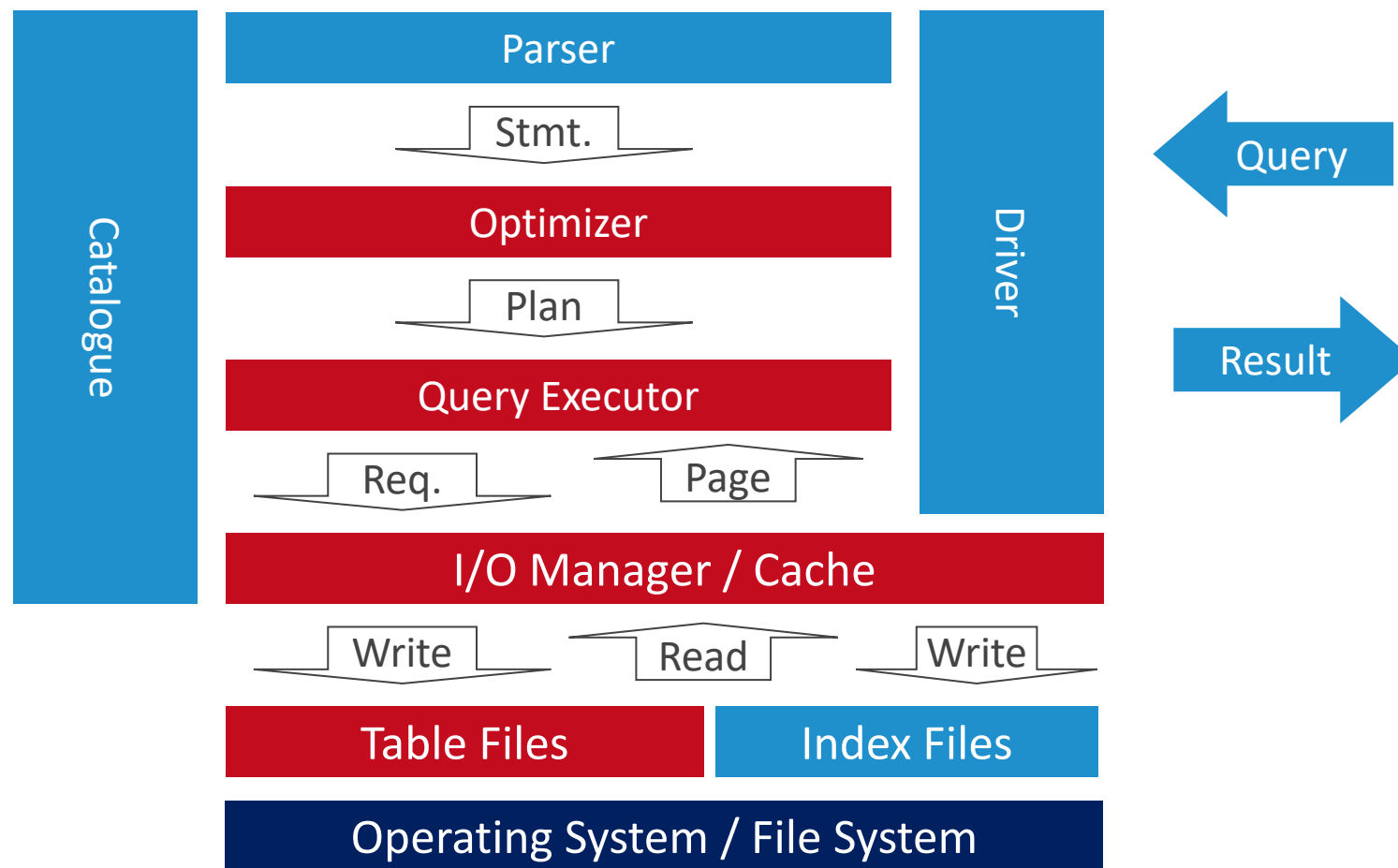


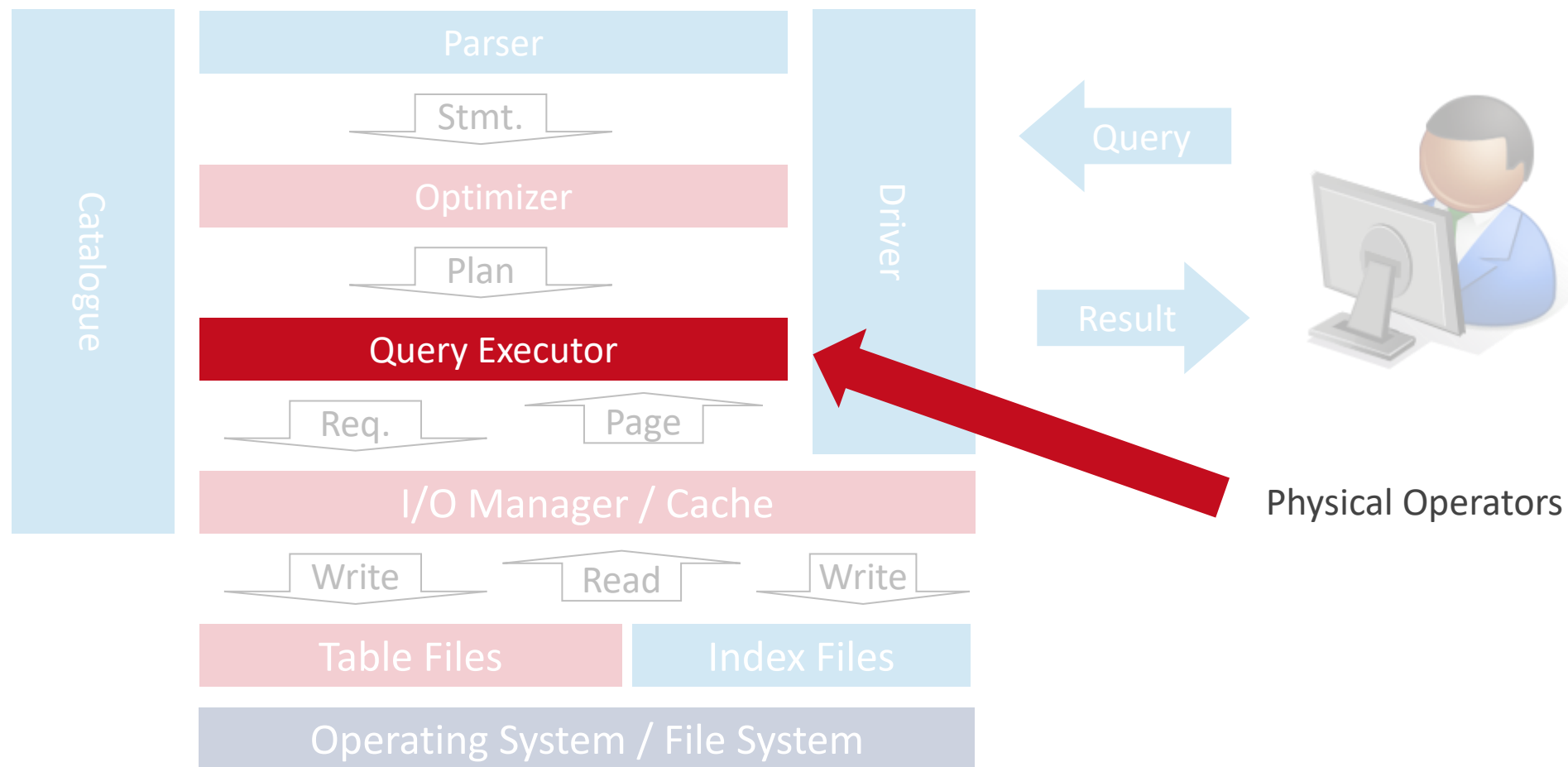
DBTLAB – Exercise 4: Query Execution - Operators I

Rudi Poepel Lemaitre
based on material from Volker Markl

Basic system architecture



Where are we today?



What we have so far

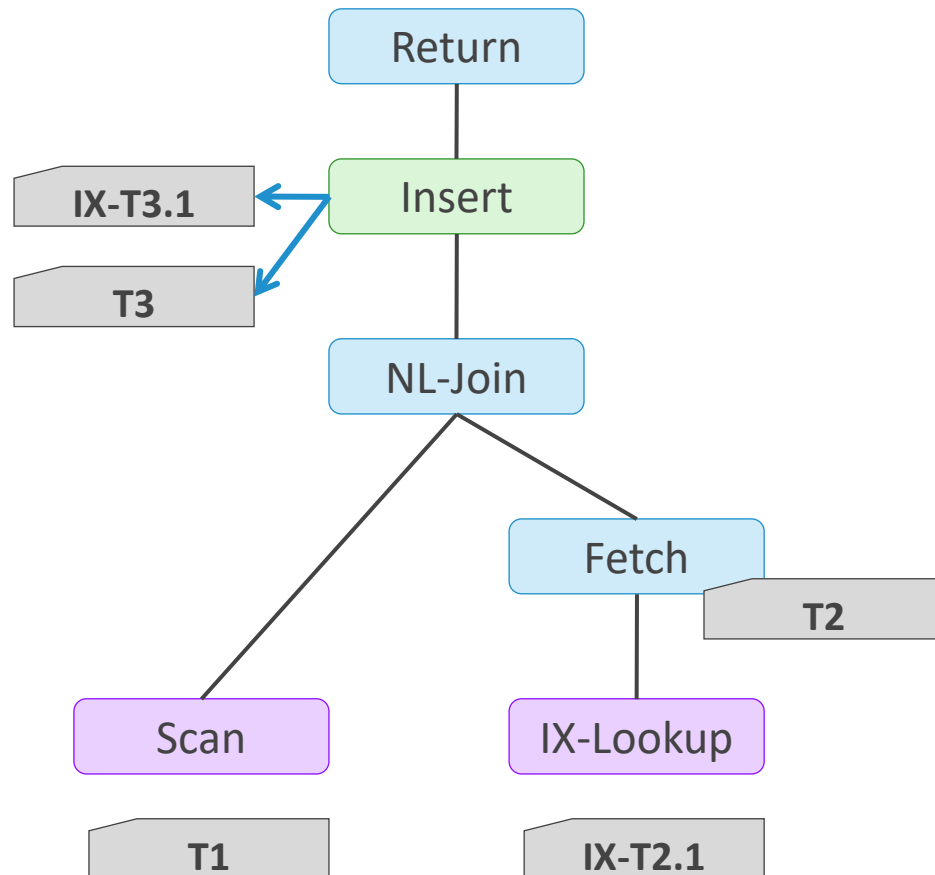
- So far, we have:
 - Files storing table data in a well accessible layout.
 - Buffers / Caches / Queues to reduce access costs to those structures.
 - ➔ I/O abstraction of the physical storage.
- Next: Operations to work with that data.

Physical Operators for our System

- Basic Data Access
 - TableScan
 - Fetch
 - IndexScan / -Lookup
- Joins
 - Nested-Loop-Join
 - Merge-Join
 - (Hybrid-Hash-Join)
- Sort
- Filter
- Group By

Correspondence to
Relational Algebra?
What is missing?

Query Execution Plans



```

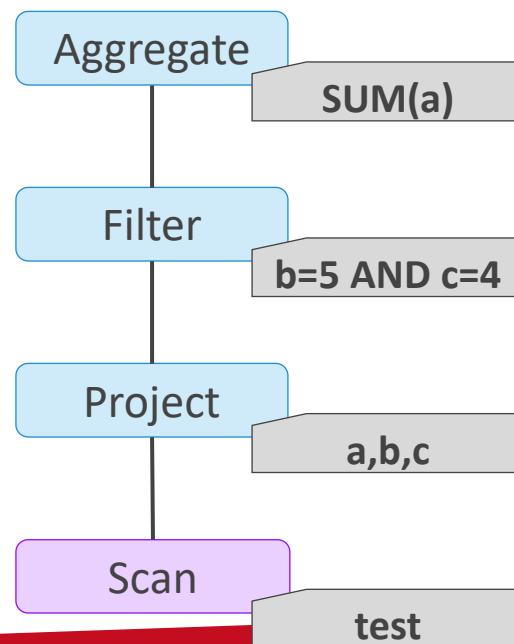
INSERT INTO T3
SELECT t1.a, t2.b
FROM tab1 t1, tab2 t2
WHERE t1.p = t2.f AND t1.num > 42
  
```

- Execution plans are composed of operators.
- Operators are small units of functionality.
- Can be composed to fully functional plans.
- Exact behavior is parameterized.
 - Predicates.
 - Which columns are propagated.
 - ...

Query Execution Models

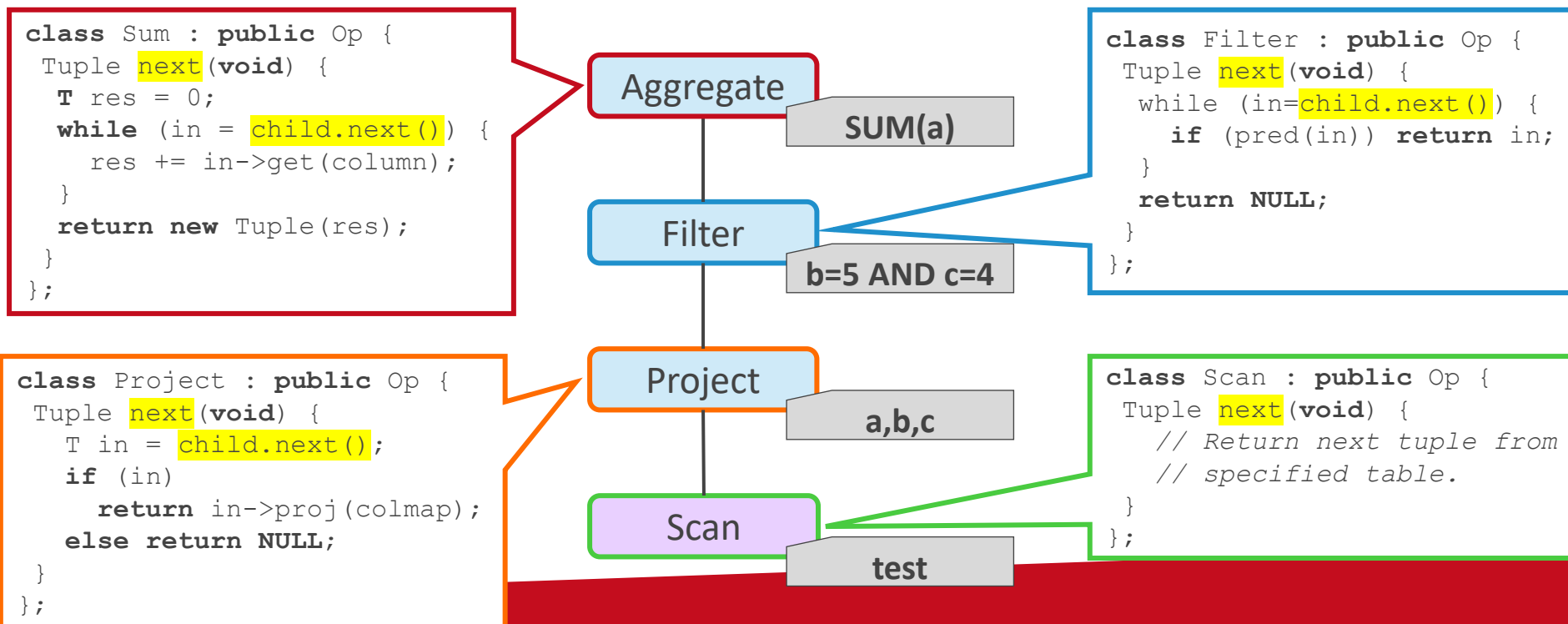
Query Execution Models

- So, how do we actually execute the query plan?
- Let us look at the following query:
`SELECT sum(a) FROM test WHERE b=5 AND c=4;`
- And the corresponding plan:



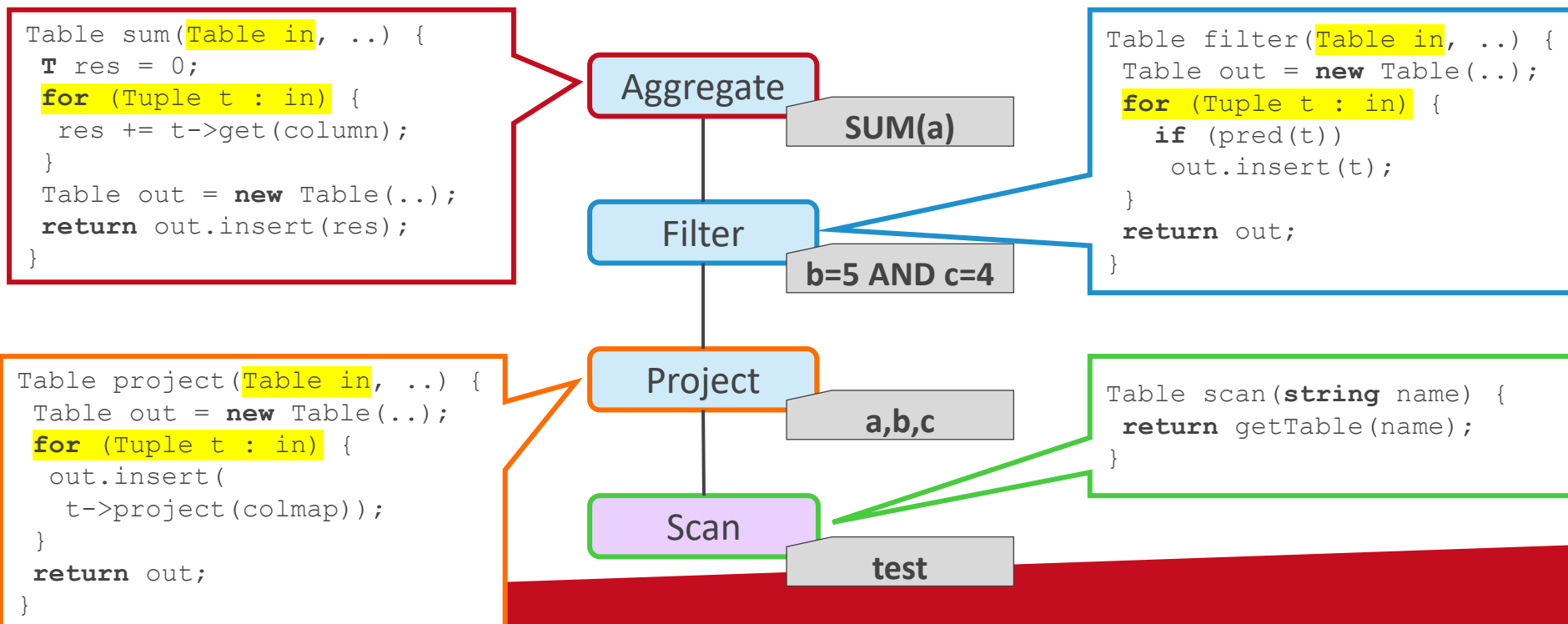
The Iterator Model

- In the iterator – or tuple-at-a-time - model, each operator returns exactly one tuple.
- To build the plan, operators are recursively chained:
Each operator directly calls its child operators to get its next input tuple.



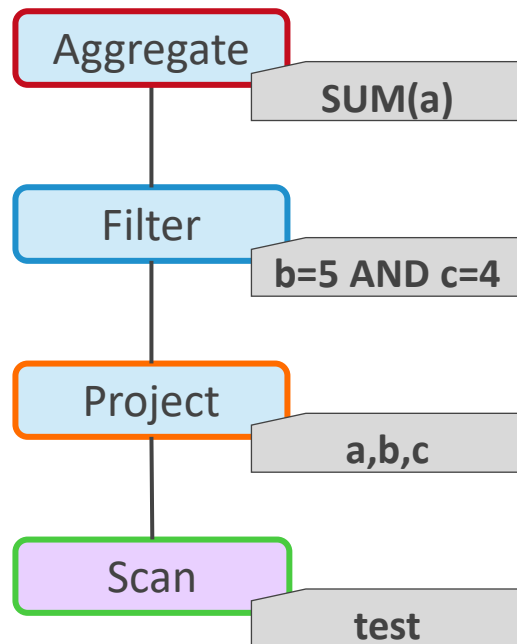
The Bulk-Processing Model

- In the bulk – or table-at-a-time – model, each operator returns one (or more) tables.
- Operators process, read and write whole tables.
- To build the plan, operators are called sequentially, passing the generated tables as input to next operators.



Dynamic Compilation

- Dynamic code compilation is used to generate custom operators at runtime.
 - Usually, these generated operators are functions that process whole tables.
 - The plan simply consists of the generated operator function.



```

Table gen_fun_1234(void) {
    Table in = new Table("test");
    T res = 0;
    for (Tuple t : test) {
        t = t->project(colmap);
        if (    t->get(b) == 5
            && t->get(c) == 4)
            res += t->get(a);
    }
    Table out = new Table(...);
    return out.insert(res);
}
  
```

Query Execution Models

- Iterator:
 - Very flexible, and easy to implement.
 - However: Extremely high CPU load (for each result tuple, at least one function call) and bad cache coherency.
 - ➔ Well-suited for disk-based database systems, not recommended for in-memory systems.
- Bulk-Processing:
 - More CPU-friendly than the iterator model (one function call per operator, good cache coherence thanks to sequential reads).
 - However: Additional I/O load, since each operator materializes its result (and multiple passes over the input might be required).
 - ➔ Well-suited for in-memory column stores.

Query Execution Models

- Dynamic Compilation:
 - Best of both worlds: Good CPU performance and much smaller I/O load than bulk/processing.
 - However: Complex to develop and maintain, adds additional one-time costs to query execution due to compilation step.
 - ➔ Well-suited for complex queries over in-memory column (or row) stores.

Our choice: The Iterator Model.

- For the operators in our mini-dbs, we use the **iterator model**.
- Reasons:
 - We are operating from disk, which means additional CPU load is negligible.
 - The iterator model is easier to implement and fits well with the object-oriented programming model in Java.
 - It makes the optimizer (slightly) easier to implement.
 - And last-but-not-least: The lecture (and the course book) from DBT only discuss the iterator model in detail.

PhysicalPlanOperator API

PhysicalPlanOperator Interface

- Following the iterator model, in our mini-dbs, each operator will implement `de.tuberlin.dima.minidb.qexec.PhysicalPlanOperator`
 - This interface requires three functions:
 - `open`
 - `next`
 - `close`

Open

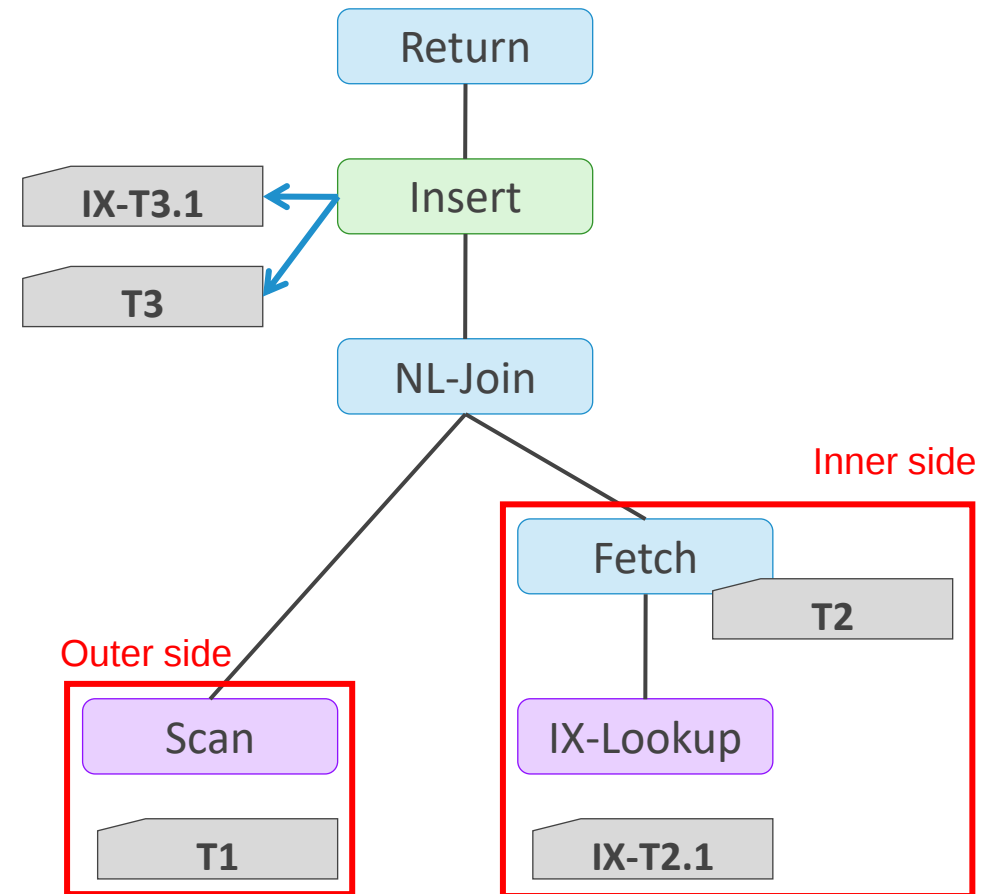
- Calls open on all children, initializes an initial state.
- Examples:
 - TableScan: Pre-fetch first set of pages.
 - IndexLookup: Descend the B⁺-Tree to find the first leaf containing relevant entries.
 - Hash-Join: Allocate and build the Hash Table from the build-side input.
- **open ()** method accepts one parameter.
 - The tuple that the operator and its sub-plan are opened for correlation.

```
public void open(DataTuple correlated) throws QueryExecutionException;
```

→ can be ignored for now, will be discussed next week. For this week assume it is always **null**, which means no correlation.

Correlated Plans Example

- Consider the sub-plan of the nested-loop-join to the right.
- The inner side is opened for every tuple from the *Table-Scan*.
 - Is opened as correlated to the current tuple from the outer loop.
 - That tuple is passed as a parameter to **open ()**.
- Each **next ()** call on the Fetch operator:
 - Calls next() on the IX-Scan.
 - IX-Scan gives next RID for tuple with the same join key as the current outer tuple (or null, if no further)
 - Fetch gets the tuple based on the RID
- Nested-Loop-Join does not even need to explicitly evaluate the join predicate
 - Join predicate is evaluated through the index lookup.



Next

- Next method returns always exactly one tuple, or null, if no more tuples are available.
- Two cases:
 - Leaf operator (TableScan, IndexScan): Access the data and produce tuples.
 - Others: Call next() on child (or children) to get an input until it is possible to produce or propagate a tuple.

Close

- Inverse of Open, releases all resources.
- Does not have any parameters.
- Examples:
 - HashJoin: De-allocate the Hash-Table
 - Sort: Release memory
 - IndexLookup: Unpin the root node

Exercise 4

Exercise 4

- Implement 2 operators using the following interfaces:
 - `de.tuberlin.dima.minidb.qexec.TableScanOperator`
 - `de.tuberlin.dima.minidb.qexec.IndexScanOperator`
- Implement the following factory methods:
 - `createTableScanOperator`
 - `createIndexScanOperator`

Table Scan Operator

- **Functionality:** Retrieve all tuples from a table and evaluate basic local predicates.
 - Performs pre-fetching of future pages.
- **Input:** None, data is directly obtained from Table.
- **Output:** Data Tuples .
- Parameters for instantiation:
 - Buffer Pool to get the pages.
 - Resource ID of the accessed table.
 - Column map with the index of the columns to project (order sensitive!).
 - Predicates to evaluate.
 - Pre-fetch length.

Hint: Look at the TablePage interface to iterate over the tuples in a page.

Example Column Map

Consider a tuple with 5 attributes A to E:

A	B	C	D	E
---	---	---	---	---

Then a column map [0,1,2] will project the following tuple:

A	B	C
---	---	---

Each column map is order-sensitive!

Consider the column map [4,3], then the correct tuple looks like this:

E	D
---	---



The following tuple is wrong, because the order is not the same as required by the column map [4,3]:

D	E
---	---



Index Scan Operator

- An index scan always produces exactly one column: The key-column.
- **Functionality:** Alternative to a TableScan
 - When only a single column (the key column) is needed.
- **Input:** None, data is directly obtained from the index.
- **Output:** Data Tuples (with a single field) .
- Parameters for instantiation:
 - B+Tree index to access.
 - Start and stop keys (predicate range).

Hint: Look at the BTreeIndex interface to get the requested keys.

Tests and points

Test name	Points	Description
testTableScan	3,5	Tests the table scan operator using different predicates and page sizes.
testIndexScan	3,5	Tests the index scan operator using different page sizes and key ranges.